**Appendix A**

**Code**

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import seaborn as sns

from datetime import datetime as dt

from zipfile import ZipFile

# Extracting the ZIP file

zip\_path = '/content/archive(6).zip'

extract\_dir = '/content/Datasets'

with ZipFile(zip\_path, 'r') as zip\_ref:

zip\_ref.extractall(extract\_dir)

print("Extraction complete.")

folder = '/content/Datasets/Datasets'

folder='/content/Datasets/Datasets'

raw\_data\_1 = pd.read\_csv('/content/Datasets/Datasets/2000-01.csv')

raw\_data\_2 = pd.read\_csv('/content/Datasets/Datasets/2001-02.csv')

raw\_data\_3 = pd.read\_csv('/content/Datasets/Datasets/2002-03.csv')

raw\_data\_4 = pd.read\_csv('/content/Datasets/Datasets/2003-04.csv')

raw\_data\_5 = pd.read\_csv('/content/Datasets/Datasets/2004-05.csv')

raw\_data\_6 = pd.read\_csv('/content/Datasets/Datasets/2005-06.csv')

raw\_data\_7 = pd.read\_csv('/content/Datasets/Datasets/2006-07.csv')

raw\_data\_8 = pd.read\_csv('/content/Datasets/Datasets/2007-08.csv')

raw\_data\_9 = pd.read\_csv('/content/Datasets/Datasets/2008-09.csv')

raw\_data\_10 = pd.read\_csv('/content/Datasets/Datasets/2009-10.csv')

raw\_data\_11 = pd.read\_csv('/content/Datasets/Datasets/2010-11.csv')

raw\_data\_12 = pd.read\_csv('/content/Datasets/Datasets/2011-12.csv')

raw\_data\_13 = pd.read\_csv('/content/Datasets/Datasets/2012-13.csv')

raw\_data\_14 = pd.read\_csv('/content/Datasets/Datasets/2013-14.csv')

raw\_data\_15 = pd.read\_csv('/content/Datasets/Datasets/2014-15.csv')

raw\_data\_16 = pd.read\_csv('/content/Datasets/Datasets/2015-16.csv')

raw\_data\_17 = pd.read\_csv('/content/Datasets/Datasets/2016-17.csv')

raw\_data\_18 = pd.read\_csv('/content/Datasets/Datasets/2017-18.csv')

# Combine the manually loaded datasets into a list

datasets = [raw\_data\_1, raw\_data\_2, raw\_data\_3, raw\_data\_4, raw\_data\_5,

raw\_data\_6, raw\_data\_7, raw\_data\_8, raw\_data\_9, raw\_data\_10,

raw\_data\_11, raw\_data\_12, raw\_data\_13, raw\_data\_14, raw\_data\_15,

raw\_data\_16, raw\_data\_17, raw\_data\_18]

seasons = [f'{year}-{str(year + 1)[-2:]}' for year in range(2000, 2018)]

for i, season in enumerate(seasons):

print(f"Data for {season}:")

display(datasets[i].head())

"""EDA"""

# Displaying data types and null values for all datasets

for i, season in enumerate(seasons):

print(f"Data Types and Null Values for {season}:")

print(datasets[i].dtypes)

print("\nMissing values:\n", datasets[i].isnull().sum())

print("\n")

# Checking unique values and duplicates

for i, season in enumerate(seasons):

print(f"Unique Values and Duplicates for {season}:")

print("Unique values:")

print(datasets[i].nunique())

print("Number of duplicates:", datasets[i].duplicated().sum())

print("\n")

# Summary statistics for numeric and categorical features

for i, season in enumerate(seasons):

print(f"Summary Statistics for {season}:")

print(datasets[i].describe(include='all'))

print("\n")

# Distribution of Goals

for i, season in enumerate(seasons):

plt.figure(figsize=(12, 6))

sns.histplot(datasets[i]['FTHG'], bins=10, kde=True, label='Home Goals', color='blue')

sns.histplot(datasets[i]['FTAG'], bins=10, kde=True, label='Away Goals', color='red')

plt.title(f'Distribution of Goals Scored - {season}')

plt.xlabel('Goals')

plt.ylabel('Frequency')

plt.legend()

plt.show()

# Boxplots for Home and Away Goals

for i, season in enumerate(seasons):

plt.figure(figsize=(10, 6))

sns.boxplot(data=datasets[i][['FTHG', 'FTAG']])

plt.title(f'Boxplot of Home and Away Goals - {season}')

plt.xlabel('Team')

plt.ylabel('Goals')

plt.xticks([0, 1], ['Home', 'Away'])

plt.show()

# Count plots for Full Time Result (FTR)

for i, season in enumerate(seasons):

plt.figure(figsize=(8, 4))

sns.countplot(datasets[i]['FTR'], palette='coolwarm')

plt.title(f'Distribution of Full Time Result (FTR) - {season}')

plt.xlabel('Result')

plt.ylabel('Count')

plt.show()

# Correlation Matrix

for i, season in enumerate(seasons):

plt.figure(figsize=(14, 10))

numeric\_data = datasets[i].select\_dtypes(include=[np.number])

correlation\_matrix = numeric\_data.corr()

sns.heatmap(correlation\_matrix, annot=True, fmt='.2f', cmap='coolwarm', linewidths=0.5)

plt.title(f'Correlation Matrix - {season}')

plt.show()

# Average goals scored by Home and Away teams

for i, season in enumerate(seasons):

home\_goals = datasets[i]['FTHG'].mean()

away\_goals = datasets[i]['FTAG'].mean()

print(f"Average Goals Scored - {season}:")

print(f"Home: {home\_goals:.2f}, Away: {away\_goals:.2f}\n")

# Calculating win rates for home, away, and draws

for i, season in enumerate(seasons):

total\_matches = datasets[i].shape[0]

home\_wins = datasets[i][datasets[i]['FTR'] == 'H'].shape[0]

away\_wins = datasets[i][datasets[i]['FTR'] == 'A'].shape[0]

draws = datasets[i][datasets[i]['FTR'] == 'D'].shape[0]

print(f"Win Rates for {season}:")

print(f"Home Win Rate: {home\_wins / total\_matches \* 100:.2f}%")

print(f"Away Win Rate: {away\_wins / total\_matches \* 100:.2f}%")

print(f"Draw Rate: {draws / total\_matches \* 100:.2f}%\n")

# Calculating goal differences for each match

for i, season in enumerate(seasons):

datasets[i]['GoalDifference'] = datasets[i]['FTHG'] - datasets[i]['FTAG']

plt.figure(figsize=(8, 4))

sns.histplot(datasets[i]['GoalDifference'], bins=20, kde=True, color='purple')

plt.title(f'Distribution of Goal Differences - {season}')

plt.xlabel('Goal Difference')

plt.ylabel('Frequency')

plt.show()

# Analysis of total goals scored over the season

total\_goals = []

seasons = [f'{year}-{str(year + 1)[-2:]}' for year in range(2000, 2018)]

for i, season in enumerate(seasons):

total\_goals.append(datasets[i]['FTHG'].sum() + datasets[i]['FTAG'].sum())

plt.figure(figsize=(12, 6))

sns.lineplot(x=seasons, y=total\_goals, marker='o')

plt.title('Total Goals Scored per Season')

plt.xlabel('Season')

plt.ylabel('Total Goals')

plt.xticks(rotation=45)

plt.show()

# Combine all season datasets into a single DataFrame

combined\_data = pd.concat(datasets, ignore\_index=True)

# Display the first few rows of the combined dataset

print("Combined DataFrame:")

display(combined\_data.head())

# Compute the correlation matrix for the combined dataset

numeric\_data\_combined = combined\_data.select\_dtypes(include=[np.number]) # Select only numeric columns

correlation\_matrix\_combined = numeric\_data\_combined.corr()

# Display the correlation matrix

print("Combined Correlation Matrix:")

display(correlation\_matrix\_combined)

# Select required columns

columns\_req = ['Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG', 'FTR']

playing\_statistics\_1 = raw\_data\_1[columns\_req]

playing\_statistics\_2 = raw\_data\_2[columns\_req]

playing\_statistics\_3 = raw\_data\_3[columns\_req]

playing\_statistics\_4 = raw\_data\_4[columns\_req]

playing\_statistics\_5 = raw\_data\_5[columns\_req]

playing\_statistics\_6 = raw\_data\_6[columns\_req]

playing\_statistics\_7 = raw\_data\_7[columns\_req]

playing\_statistics\_8 = raw\_data\_8[columns\_req]

playing\_statistics\_9 = raw\_data\_9[columns\_req]

playing\_statistics\_10 = raw\_data\_10[columns\_req]

playing\_statistics\_11 = raw\_data\_11[columns\_req]

playing\_statistics\_12 = raw\_data\_12[columns\_req]

playing\_statistics\_13 = raw\_data\_13[columns\_req]

playing\_statistics\_14 = raw\_data\_14[columns\_req]

playing\_statistics\_15 = raw\_data\_15[columns\_req]

playing\_statistics\_16 = raw\_data\_16[columns\_req]

playing\_statistics\_17 = raw\_data\_17[columns\_req]

playing\_statistics\_18 = raw\_data\_18[columns\_req]

# Concatenate all seasons' data

all\_data = pd.concat([playing\_statistics\_1, playing\_statistics\_2, playing\_statistics\_3,

playing\_statistics\_4, playing\_statistics\_5, playing\_statistics\_6,

playing\_statistics\_7, playing\_statistics\_8, playing\_statistics\_9,

playing\_statistics\_10, playing\_statistics\_11, playing\_statistics\_12,

playing\_statistics\_13, playing\_statistics\_14, playing\_statistics\_15,

playing\_statistics\_16, playing\_statistics\_17, playing\_statistics\_18],

ignore\_index=True)

# Display the first few rows

print(all\_data.head())

# Additional EDA Steps

# 1. Distribution of Full Time Results (FTR)

sns.countplot(x='FTR', data=all\_data)

plt.title('Distribution of Full Time Results')

plt.show()

# 2. Goals scored by Home vs Away teams

sns.histplot(all\_data['FTHG'], bins=20, color='blue', label='Home Goals', kde=True)

sns.histplot(all\_data['FTAG'], bins=20, color='red', label='Away Goals', kde=True)

plt.legend()

plt.title('Distribution of Goals Scored')

plt.show()

# Select only numeric columns for correlation matrix calculation

numeric\_cols = all\_data.select\_dtypes(include=[np.number]).columns

corr = all\_data[numeric\_cols].corr()

# Plot the heatmap

sns.heatmap(corr, annot=True, fmt=".2f", cmap='coolwarm')

plt.title('Correlation Matrix')

plt.show()

# Additional visualizations and EDA

# Distribution of goals scored by home and away teams

sns.histplot(playing\_stat['HTGS'], bins=20, kde=True, color='blue', label='Home Team Goals')

sns.histplot(playing\_stat['ATGS'], bins=20, kde=True, color='red', label='Away Team Goals')

plt.legend()

plt.title('Distribution of Goals Scored')

plt.show()

# Correlation Heatmap of all features

numeric\_features = playing\_stat.select\_dtypes(include=[np.number])

corr\_matrix = numeric\_features.corr()

plt.figure(figsize=(12, 8))

sns.heatmap(corr\_matrix, annot=True, fmt='.2f', cmap='coolwarm', linewidths=0.5)

plt.title('Correlation Matrix of Features')

plt.show()

"""Feature Engineering"""

def get\_goals\_scored(playing\_stat):

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

for i in range(len(playing\_stat)):

HTGS = playing\_stat.iloc[i]['FTHG']

ATGS = playing\_stat.iloc[i]['FTAG']

teams[playing\_stat.iloc[i].HomeTeam].append(HTGS)

teams[playing\_stat.iloc[i].AwayTeam].append(ATGS)

num\_matchweeks = max(len(v) for v in teams.values())

GoalsScored = pd.DataFrame(data=teams, index=range(1, num\_matchweeks + 1)).T

GoalsScored[0] = 0

for i in range(2, num\_matchweeks + 1):

GoalsScored[i] = GoalsScored[i] + GoalsScored[i-1]

return GoalsScored

def get\_goals\_conceded(playing\_stat):

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

for i in range(len(playing\_stat)):

ATGC = playing\_stat.iloc[i]['FTHG']

HTGC = playing\_stat.iloc[i]['FTAG']

teams[playing\_stat.iloc[i].HomeTeam].append(HTGC)

teams[playing\_stat.iloc[i].AwayTeam].append(ATGC)

num\_matchweeks = max(len(v) for v in teams.values())

GoalsConceded = pd.DataFrame(data=teams, index=range(1, num\_matchweeks + 1)).T

GoalsConceded[0] = 0

for i in range(2, num\_matchweeks + 1):

GoalsConceded[i] = GoalsConceded[i] + GoalsConceded[i-1]

return GoalsConceded

def get\_gss(playing\_stat):

GC = get\_goals\_conceded(playing\_stat)

GS = get\_goals\_scored(playing\_stat)

j = 0

HTGS = []

ATGS = []

HTGC = []

ATGC = []

num\_games = len(playing\_stat)

num\_matchweeks = min(GS.shape[1], GC.shape[1])

for i in range(num\_games):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

HTGS.append(GS.loc[ht].iloc[j])

ATGS.append(GS.loc[at].iloc[j])

HTGC.append(GC.loc[ht].iloc[j])

ATGC.append(GC.loc[at].iloc[j])

if (i + 1) % 10 == 0 and j < num\_matchweeks - 1:

j += 1

playing\_stat['HTGS'] = HTGS

playing\_stat['ATGS'] = ATGS

playing\_stat['HTGC'] = HTGC

playing\_stat['ATGC'] = ATGC

return playing\_stat

# Apply the feature engineering functions to each dataset

playing\_statistics\_1 = get\_gss(playing\_statistics\_1)

playing\_statistics\_2 = get\_gss(playing\_statistics\_2)

playing\_statistics\_3 = get\_gss(playing\_statistics\_3)

playing\_statistics\_4 = get\_gss(playing\_statistics\_4)

playing\_statistics\_5 = get\_gss(playing\_statistics\_5)

playing\_statistics\_6 = get\_gss(playing\_statistics\_6)

playing\_statistics\_7 = get\_gss(playing\_statistics\_7)

playing\_statistics\_8 = get\_gss(playing\_statistics\_8)

playing\_statistics\_9 = get\_gss(playing\_statistics\_9)

playing\_statistics\_10 = get\_gss(playing\_statistics\_10)

playing\_statistics\_11 = get\_gss(playing\_statistics\_11)

playing\_statistics\_12 = get\_gss(playing\_statistics\_12)

playing\_statistics\_13 = get\_gss(playing\_statistics\_13)

playing\_statistics\_14 = get\_gss(playing\_statistics\_14)

playing\_statistics\_15 = get\_gss(playing\_statistics\_15)

playing\_statistics\_16 = get\_gss(playing\_statistics\_16)

playing\_statistics\_17 = get\_gss(playing\_statistics\_17)

playing\_statistics\_18 = get\_gss(playing\_statistics\_18)

import pandas as pd

# Assuming 'playing\_stat' is the DataFrame for the match statistics

def get\_matchres(playing\_stat):

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

for i in range(len(playing\_stat)):

if playing\_stat.iloc[i].FTR == 'H':

teams[playing\_stat.iloc[i].HomeTeam].append('W')

teams[playing\_stat.iloc[i].AwayTeam].append('L')

elif playing\_stat.iloc[i].FTR == 'A':

teams[playing\_stat.iloc[i].AwayTeam].append('W')

teams[playing\_stat.iloc[i].HomeTeam].append('L')

else:

teams[playing\_stat.iloc[i].AwayTeam].append('D')

teams[playing\_stat.iloc[i].HomeTeam].append('D')

return pd.DataFrame(data=teams, index=range(1, 39)).T

def get\_points(result):

if result == 'W':

return 3

elif result == 'D':

return 1

else:

return 0

def get\_cuml\_points(matchres):

matchres\_points = matchres.applymap(get\_points)

num\_matchweeks = matchres\_points.shape[1]

matchres\_points.insert(0, 0, 0)

for i in range(2, num\_matchweeks + 1):

matchres\_points[i] = matchres\_points[i] + matchres\_points[i - 1]

return matchres\_points

def get\_agg\_points(playing\_stat):

matchres = get\_matchres(playing\_stat)

cum\_pts = get\_cuml\_points(matchres)

HTP = []

ATP = []

j = 0

num\_games = len(playing\_stat)

num\_matchweeks = cum\_pts.shape[1]

for i in range(num\_games):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

if j < num\_matchweeks:

HTP.append(cum\_pts.loc[ht].iloc[j])

ATP.append(cum\_pts.loc[at].iloc[j])

if (i + 1) % 10 == 0 and j < num\_matchweeks - 1:

j += 1

playing\_stat['HTP'] = HTP

playing\_stat['ATP'] = ATP

return playing\_stat

def get\_form(playing\_stat, num):

matchres = get\_matchres(playing\_stat)

form\_final = matchres.copy()

for i in range(num, 39):

form\_final[i] = ''

for j in range(num):

form\_final[i] += matchres[i-j]

return form\_final

def add\_form(playing\_stat, num):

form = get\_form(playing\_stat, num)

h = ['M' for \_ in range(num \* 10)]

a = ['M' for \_ in range(num \* 10)]

j = num

for i in range(num \* 10, len(playing\_stat)):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

past = form.loc[ht][j]

h.append(past[num-1])

past = form.loc[at][j]

a.append(past[num-1])

if ((i + 1) % 10) == 0:

j += 1

playing\_stat[f'HM{num}'] = h

playing\_stat[f'AM{num}'] = a

return playing\_stat

def add\_form\_df(playing\_statistics):

for num in range(1, 6):

playing\_statistics = [add\_form(df, num) for df in playing\_statistics]

return playing\_statistics

# Apply cumulative points calculation

playing\_statistics\_1 = get\_agg\_points(playing\_statistics\_1)

playing\_statistics\_2 = get\_agg\_points(playing\_statistics\_2)

playing\_statistics\_3 = get\_agg\_points(playing\_statistics\_3)

playing\_statistics\_4 = get\_agg\_points(playing\_statistics\_4)

playing\_statistics\_5 = get\_agg\_points(playing\_statistics\_5)

playing\_statistics\_6 = get\_agg\_points(playing\_statistics\_6)

playing\_statistics\_7 = get\_agg\_points(playing\_statistics\_7)

playing\_statistics\_8 = get\_agg\_points(playing\_statistics\_8)

playing\_statistics\_9 = get\_agg\_points(playing\_statistics\_9)

playing\_statistics\_10 = get\_agg\_points(playing\_statistics\_10)

playing\_statistics\_11 = get\_agg\_points(playing\_statistics\_11)

playing\_statistics\_12 = get\_agg\_points(playing\_statistics\_12)

playing\_statistics\_13 = get\_agg\_points(playing\_statistics\_13)

playing\_statistics\_14 = get\_agg\_points(playing\_statistics\_14)

playing\_statistics\_15 = get\_agg\_points(playing\_statistics\_15)

playing\_statistics\_16 = get\_agg\_points(playing\_statistics\_16)

playing\_statistics\_17 = get\_agg\_points(playing\_statistics\_17)

playing\_statistics\_18 = get\_agg\_points(playing\_statistics\_18)

# Adding form (last n matches) data

playing\_statistics\_1 = add\_form\_df([playing\_statistics\_1])[0]

playing\_statistics\_2 = add\_form\_df([playing\_statistics\_2])[0]

playing\_statistics\_3 = add\_form\_df([playing\_statistics\_3])[0]

playing\_statistics\_4 = add\_form\_df([playing\_statistics\_4])[0]

playing\_statistics\_5 = add\_form\_df([playing\_statistics\_5])[0]

playing\_statistics\_6 = add\_form\_df([playing\_statistics\_6])[0]

playing\_statistics\_7 = add\_form\_df([playing\_statistics\_7])[0]

playing\_statistics\_8 = add\_form\_df([playing\_statistics\_8])[0]

playing\_statistics\_9 = add\_form\_df([playing\_statistics\_9])[0]

playing\_statistics\_10 = add\_form\_df([playing\_statistics\_10])[0]

playing\_statistics\_11 = add\_form\_df([playing\_statistics\_11])[0]

playing\_statistics\_12 = add\_form\_df([playing\_statistics\_12])[0]

playing\_statistics\_13 = add\_form\_df([playing\_statistics\_13])[0]

playing\_statistics\_14 = add\_form\_df([playing\_statistics\_14])[0]

playing\_statistics\_15 = add\_form\_df([playing\_statistics\_15])[0]

playing\_statistics\_16 = add\_form\_df([playing\_statistics\_16])[0]

playing\_statistics\_17 = add\_form\_df([playing\_statistics\_17])[0]

playing\_statistics\_18 = add\_form\_df([playing\_statistics\_18])[0]

# Preparing the final dataset by concatenating all seasons' data

playing\_stat = pd.concat([

playing\_statistics\_1, playing\_statistics\_2, playing\_statistics\_3,

playing\_statistics\_4, playing\_statistics\_5, playing\_statistics\_6,

playing\_statistics\_7, playing\_statistics\_8, playing\_statistics\_9,

playing\_statistics\_10, playing\_statistics\_11, playing\_statistics\_12,

playing\_statistics\_13, playing\_statistics\_14, playing\_statistics\_15,

playing\_statistics\_16, playing\_statistics\_17, playing\_statistics\_18

], ignore\_index=True)

# Check if the columns HTFormPts and ATFormPts exist, if not, calculate them

if 'HTFormPts' not in playing\_stat.columns or 'ATFormPts' not in playing\_stat.columns:

# Assuming HTFormPtsStr and ATFormPtsStr columns exist

playing\_stat['HTFormPtsStr'] = (

playing\_stat['HM1'] + playing\_stat['HM2'] + playing\_stat['HM3'] + playing\_stat['HM4'] + playing\_stat['HM5']

)

playing\_stat['ATFormPtsStr'] = (

playing\_stat['AM1'] + playing\_stat['AM2'] + playing\_stat['AM3'] + playing\_stat['AM4'] + playing\_stat['AM5']

)

def get\_form\_points(string):

sum\_points = 0

for letter in string:

sum\_points += get\_points(letter)

return sum\_points

playing\_stat['HTFormPts'] = playing\_stat['HTFormPtsStr'].apply(get\_form\_points)

playing\_stat['ATFormPts'] = playing\_stat['ATFormPtsStr'].apply(get\_form\_points)

# Calculating additional features

playing\_stat['HTGD'] = playing\_stat['HTGS'] - playing\_stat['HTGC']

playing\_stat['ATGD'] = playing\_stat['ATGS'] - playing\_stat['ATGC']

playing\_stat['DiffPts'] = playing\_stat['HTP'] - playing\_stat['ATP']

playing\_stat['DiffFormPts'] = playing\_stat['HTFormPts'] - playing\_stat['ATFormPts']

# Ensure that MW column is correctly calculated and exists

if 'MW' not in playing\_stat.columns:

def get\_mw(df):

j = 1

MatchWeek = []

for i in range(len(df)):

MatchWeek.append(j)

if ((i + 1) % 10) == 0:

j += 1

df['MW'] = MatchWeek

return df

playing\_stat = get\_mw(playing\_stat)

# Scaling certain features by Matchweek

playing\_stat['MW'] = playing\_stat['MW'].astype(float)

cols\_to\_scale = ['HTGD', 'ATGD', 'DiffPts', 'DiffFormPts', 'HTP', 'ATP']

for col in cols\_to\_scale:

playing\_stat[col] = playing\_stat[col] / playing\_stat['MW']

# Transform the target variable (FTR) to binary classification ('H' or 'NH')

playing\_stat['FTR'] = playing\_stat['FTR'].apply(lambda x: 'H' if x == 'H' else 'NH')

# Saving the final dataset

playing\_stat.to\_csv('/content/final\_dataset.csv', index=False)

# Loading the dataset for further use

dataset = pd.read\_csv('/content/final\_dataset.csv')

columns\_req = ['Date','HomeTeam','AwayTeam','FTHG','FTAG','FTR']

playing\_statistics\_1 = raw\_data\_1[columns\_req]

playing\_statistics\_2 = raw\_data\_2[columns\_req]

playing\_statistics\_3 = raw\_data\_3[columns\_req]

playing\_statistics\_4 = raw\_data\_4[columns\_req]

playing\_statistics\_5 = raw\_data\_5[columns\_req]

playing\_statistics\_6 = raw\_data\_6[columns\_req]

playing\_statistics\_7 = raw\_data\_7[columns\_req]

playing\_statistics\_8 = raw\_data\_8[columns\_req]

playing\_statistics\_9 = raw\_data\_9[columns\_req]

playing\_statistics\_10 = raw\_data\_10[columns\_req]

playing\_statistics\_11 = raw\_data\_11[columns\_req]

playing\_statistics\_12 = raw\_data\_12[columns\_req]

playing\_statistics\_13 = raw\_data\_13[columns\_req]

playing\_statistics\_14 = raw\_data\_14[columns\_req]

playing\_statistics\_15 = raw\_data\_15[columns\_req]

playing\_statistics\_16 = raw\_data\_16[columns\_req]

playing\_statistics\_17 = raw\_data\_17[columns\_req]

playing\_statistics\_18 = raw\_data\_18[columns\_req]

import pandas as pd

def get\_goals\_scored(playing\_stat):

# Create a dictionary with team names as keys

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

# Fill in the goals scored for each team

for i in range(len(playing\_stat)):

HTGS = playing\_stat.iloc[i]['FTHG']

ATGS = playing\_stat.iloc[i]['FTAG']

teams[playing\_stat.iloc[i].HomeTeam].append(HTGS)

teams[playing\_stat.iloc[i].AwayTeam].append(ATGS)

# Determine the number of matchweeks

num\_matchweeks = max(len(v) for v in teams.values())

# Create a dataframe for goals scored where rows are teams and columns are matchweek.

GoalsScored = pd.DataFrame(data=teams, index=range(1, num\_matchweeks + 1)).T

GoalsScored[0] = 0 # Initialize the cumulative sum

# Aggregate to get cumulative goals up to that point

for i in range(2, num\_matchweeks + 1):

GoalsScored[i] = GoalsScored[i] + GoalsScored[i-1]

return GoalsScored

def get\_goals\_conceded(playing\_stat):

# Create a dictionary with team names as keys

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

# Fill in the goals conceded for each team

for i in range(len(playing\_stat)):

ATGC = playing\_stat.iloc[i]['FTHG']

HTGC = playing\_stat.iloc[i]['FTAG']

teams[playing\_stat.iloc[i].HomeTeam].append(HTGC)

teams[playing\_stat.iloc[i].AwayTeam].append(ATGC)

# Determine the number of matchweeks

num\_matchweeks = max(len(v) for v in teams.values())

# Create a dataframe for goals conceded where rows are teams and columns are matchweek.

GoalsConceded = pd.DataFrame(data=teams, index=range(1, num\_matchweeks + 1)).T

GoalsConceded[0] = 0 # Initialize the cumulative sum

# Aggregate to get cumulative goals conceded up to that point

for i in range(2, num\_matchweeks + 1):

GoalsConceded[i] = GoalsConceded[i] + GoalsConceded[i-1]

return GoalsConceded

def get\_gss(playing\_stat):

GC = get\_goals\_conceded(playing\_stat)

GS = get\_goals\_scored(playing\_stat)

j = 0

HTGS = []

ATGS = []

HTGC = []

ATGC = []

num\_games = len(playing\_stat)

num\_matchweeks = min(GS.shape[1], GC.shape[1])

for i in range(num\_games):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

HTGS.append(GS.loc[ht].iloc[j])

ATGS.append(GS.loc[at].iloc[j])

HTGC.append(GC.loc[ht].iloc[j])

ATGC.append(GC.loc[at].iloc[j])

if (i + 1) % 10 == 0 and j < num\_matchweeks - 1:

j += 1

playing\_stat['HTGS'] = HTGS

playing\_stat['ATGS'] = ATGS

playing\_stat['HTGC'] = HTGC

playing\_stat['ATGC'] = ATGC

return playing\_stat

# Apply to each dataset

playing\_statistics\_1 = get\_gss(playing\_statistics\_1)

playing\_statistics\_2 = get\_gss(playing\_statistics\_2)

playing\_statistics\_3 = get\_gss(playing\_statistics\_3)

playing\_statistics\_4 = get\_gss(playing\_statistics\_4)

playing\_statistics\_5 = get\_gss(playing\_statistics\_5)

playing\_statistics\_6 = get\_gss(playing\_statistics\_6)

playing\_statistics\_7 = get\_gss(playing\_statistics\_7)

playing\_statistics\_8 = get\_gss(playing\_statistics\_8)

playing\_statistics\_9 = get\_gss(playing\_statistics\_9)

playing\_statistics\_10 = get\_gss(playing\_statistics\_10)

playing\_statistics\_11 = get\_gss(playing\_statistics\_11)

playing\_statistics\_12 = get\_gss(playing\_statistics\_12)

playing\_statistics\_13 = get\_gss(playing\_statistics\_13)

playing\_statistics\_14 = get\_gss(playing\_statistics\_14)

playing\_statistics\_15 = get\_gss(playing\_statistics\_15)

playing\_statistics\_16 = get\_gss(playing\_statistics\_16)

playing\_statistics\_17 = get\_gss(playing\_statistics\_17)

playing\_statistics\_18 = get\_gss(playing\_statistics\_18)

import pandas as pd

def get\_points(result):

if result == 'W':

return 3

elif result == 'D':

return 1

else:

return 0

def get\_cuml\_points(matchres):

# Convert match results to points

matchres\_points = matchres.applymap(get\_points)

# Ensure there's no issue with index when initializing the cumulative points

num\_matchweeks = matchres\_points.shape[1]

num\_teams = matchres\_points.shape[0]

matchres\_points[0] = 0 # Initialize the cumulative sum with zero points

for i in range(2, num\_matchweeks + 1):

matchres\_points[i] = matchres\_points[i] + matchres\_points[i - 1]

return matchres\_points

def get\_matchres(playing\_stat):

# Create a dictionary with team names as keys

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

# the value corresponding to keys is a list containing the match result

for i in range(len(playing\_stat)):

if playing\_stat.iloc[i].FTR == 'H':

teams[playing\_stat.iloc[i].HomeTeam].append('W')

teams[playing\_stat.iloc[i].AwayTeam].append('L')

elif playing\_stat.iloc[i].FTR == 'A':

teams[playing\_stat.iloc[i].AwayTeam].append('W')

teams[playing\_stat.iloc[i].HomeTeam].append('L')

else:

teams[playing\_stat.iloc[i].AwayTeam].append('D')

teams[playing\_stat.iloc[i].HomeTeam].append('D')

# Determine the number of matchweeks based on the length of the values

num\_matchweeks = max(len(results) for results in teams.values())

# Create a DataFrame with the results and ensure correct indexing

return pd.DataFrame(data=teams, index=range(1, num\_matchweeks + 1)).T

def get\_agg\_points(playing\_stat):

matchres = get\_matchres(playing\_stat)

cum\_pts = get\_cuml\_points(matchres)

HTP = []

ATP = []

j = 0

num\_games = len(playing\_stat)

num\_matchweeks = cum\_pts.shape[1]

for i in range(num\_games):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

if j < num\_matchweeks:

HTP.append(cum\_pts.loc[ht].iloc[j])

ATP.append(cum\_pts.loc[at].iloc[j])

if (i + 1) % 10 == 0 and j < num\_matchweeks - 1:

j += 1

playing\_stat['HTP'] = HTP

playing\_stat['ATP'] = ATP

return playing\_stat

# Apply to each dataset

playing\_statistics\_1 = get\_gss(playing\_statistics\_1)

playing\_statistics\_2 = get\_gss(playing\_statistics\_2)

playing\_statistics\_3 = get\_gss(playing\_statistics\_3)

playing\_statistics\_4 = get\_gss(playing\_statistics\_4)

playing\_statistics\_5 = get\_gss(playing\_statistics\_5)

playing\_statistics\_6 = get\_gss(playing\_statistics\_6)

playing\_statistics\_7 = get\_gss(playing\_statistics\_7)

playing\_statistics\_8 = get\_gss(playing\_statistics\_8)

playing\_statistics\_9 = get\_gss(playing\_statistics\_9)

playing\_statistics\_10 = get\_gss(playing\_statistics\_10)

playing\_statistics\_11 = get\_gss(playing\_statistics\_11)

playing\_statistics\_12 = get\_gss(playing\_statistics\_12)

playing\_statistics\_13 = get\_gss(playing\_statistics\_13)

playing\_statistics\_14 = get\_gss(playing\_statistics\_14)

playing\_statistics\_15 = get\_gss(playing\_statistics\_15)

playing\_statistics\_16 = get\_gss(playing\_statistics\_16)

playing\_statistics\_17 = get\_gss(playing\_statistics\_17)

playing\_statistics\_18 = get\_gss(playing\_statistics\_18)

def get\_form(playing\_stat,num):

form = get\_matchres(playing\_stat)

form\_final = form.copy()

for i in range(num,39):

form\_final[i] = ''

j = 0

while j < num:

form\_final[i] += form[i-j]

j += 1

return form\_final

def add\_form(playing\_stat,num):

form = get\_form(playing\_stat,num)

h = ['M' for i in range(num \* 10)] # since form is not available for n MW (n\*10)

a = ['M' for i in range(num \* 10)]

j = num

for i in range((num\*10),380):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

past = form.loc[ht][j] # get past n results

h.append(past[num-1]) # 0 index is most recent

past = form.loc[at][j] # get past n results.

a.append(past[num-1]) # 0 index is most recent

if ((i + 1)% 10) == 0:

j = j + 1

playing\_stat['HM' + str(num)] = h

playing\_stat['AM' + str(num)] = a

return playing\_stat

def add\_form\_df(playing\_statistics):

playing\_statistics = add\_form(playing\_statistics,1)

playing\_statistics = add\_form(playing\_statistics,2)

playing\_statistics = add\_form(playing\_statistics,3)

playing\_statistics = add\_form(playing\_statistics,4)

playing\_statistics = add\_form(playing\_statistics,5)

return playing\_statistics

# Make changes to df

playing\_statistics\_1 = add\_form\_df(playing\_statistics\_1)

playing\_statistics\_2 = add\_form\_df(playing\_statistics\_2)

playing\_statistics\_3 = add\_form\_df(playing\_statistics\_3)

playing\_statistics\_4 = add\_form\_df(playing\_statistics\_4)

playing\_statistics\_5 = add\_form\_df(playing\_statistics\_5)

playing\_statistics\_6 = add\_form\_df(playing\_statistics\_6)

playing\_statistics\_7 = add\_form\_df(playing\_statistics\_7)

playing\_statistics\_8 = add\_form\_df(playing\_statistics\_8)

playing\_statistics\_9 = add\_form\_df(playing\_statistics\_9)

playing\_statistics\_10 = add\_form\_df(playing\_statistics\_10)

playing\_statistics\_11 = add\_form\_df(playing\_statistics\_11)

playing\_statistics\_12 = add\_form\_df(playing\_statistics\_12)

playing\_statistics\_13 = add\_form\_df(playing\_statistics\_13)

playing\_statistics\_14 = add\_form\_df(playing\_statistics\_14)

playing\_statistics\_15 = add\_form\_df(playing\_statistics\_15)

playing\_statistics\_16 = add\_form\_df(playing\_statistics\_16)

playing\_statistics\_17 = add\_form\_df(playing\_statistics\_17)

playing\_statistics\_18 = add\_form\_df(playing\_statistics\_18)

def get\_form(playing\_stat,num):

form = get\_matchres(playing\_stat)

form\_final = form.copy()

for i in range(num,39):

form\_final[i] = ''

j = 0

while j < num:

form\_final[i] += form[i-j]

j += 1

return form\_final

def add\_form(playing\_stat,num):

form = get\_form(playing\_stat,num)

h = ['M' for i in range(num \* 10)] # since form is not available for n MW (n\*10)

a = ['M' for i in range(num \* 10)]

j = num

for i in range((num\*10),380):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

past = form.loc[ht][j] # get past n results

h.append(past[num-1]) # 0 index is most recent

past = form.loc[at][j] # get past n results.

a.append(past[num-1]) # 0 index is most recent

if ((i + 1)% 10) == 0:

j = j + 1

playing\_stat['HM' + str(num)] = h

playing\_stat['AM' + str(num)] = a

return playing\_stat

def add\_form\_df(playing\_statistics):

playing\_statistics = add\_form(playing\_statistics,1)

playing\_statistics = add\_form(playing\_statistics,2)

playing\_statistics = add\_form(playing\_statistics,3)

playing\_statistics = add\_form(playing\_statistics,4)

playing\_statistics = add\_form(playing\_statistics,5)

return playing\_statistics

# Make changes to df

playing\_statistics\_1 = add\_form\_df(playing\_statistics\_1)

playing\_statistics\_2 = add\_form\_df(playing\_statistics\_2)

playing\_statistics\_3 = add\_form\_df(playing\_statistics\_3)

playing\_statistics\_4 = add\_form\_df(playing\_statistics\_4)

playing\_statistics\_5 = add\_form\_df(playing\_statistics\_5)

playing\_statistics\_6 = add\_form\_df(playing\_statistics\_6)

playing\_statistics\_7 = add\_form\_df(playing\_statistics\_7)

playing\_statistics\_8 = add\_form\_df(playing\_statistics\_8)

playing\_statistics\_9 = add\_form\_df(playing\_statistics\_9)

playing\_statistics\_10 = add\_form\_df(playing\_statistics\_10)

playing\_statistics\_11 = add\_form\_df(playing\_statistics\_11)

playing\_statistics\_12 = add\_form\_df(playing\_statistics\_12)

playing\_statistics\_13 = add\_form\_df(playing\_statistics\_13)

playing\_statistics\_14 = add\_form\_df(playing\_statistics\_14)

playing\_statistics\_15 = add\_form\_df(playing\_statistics\_15)

playing\_statistics\_16 = add\_form\_df(playing\_statistics\_16)

playing\_statistics\_17 = add\_form\_df(playing\_statistics\_17)

playing\_statistics\_18 = add\_form\_df(playing\_statistics\_18)

def get\_form(playing\_stat,num):

form = get\_matchres(playing\_stat)

form\_final = form.copy()

for i in range(num,39):

form\_final[i] = ''

j = 0

while j < num:

form\_final[i] += form[i-j]

j += 1

return form\_final

def add\_form(playing\_stat,num):

form = get\_form(playing\_stat,num)

h = ['M' for i in range(num \* 10)] # since form is not available for n MW (n\*10)

a = ['M' for i in range(num \* 10)]

j = num

for i in range((num\*10),380):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

past = form.loc[ht][j] # get past n results

h.append(past[num-1]) # 0 index is most recent

past = form.loc[at][j] # get past n results.

a.append(past[num-1]) # 0 index is most recent

if ((i + 1)% 10) == 0:

j = j + 1

playing\_stat['HM' + str(num)] = h

playing\_stat['AM' + str(num)] = a

return playing\_stat

def add\_form\_df(playing\_statistics):

playing\_statistics = add\_form(playing\_statistics,1)

playing\_statistics = add\_form(playing\_statistics,2)

playing\_statistics = add\_form(playing\_statistics,3)

playing\_statistics = add\_form(playing\_statistics,4)

playing\_statistics = add\_form(playing\_statistics,5)

return playing\_statistics

# Make changes to df

playing\_statistics\_1 = add\_form\_df(playing\_statistics\_1)

playing\_statistics\_2 = add\_form\_df(playing\_statistics\_2)

playing\_statistics\_3 = add\_form\_df(playing\_statistics\_3)

playing\_statistics\_4 = add\_form\_df(playing\_statistics\_4)

playing\_statistics\_5 = add\_form\_df(playing\_statistics\_5)

playing\_statistics\_6 = add\_form\_df(playing\_statistics\_6)

playing\_statistics\_7 = add\_form\_df(playing\_statistics\_7)

playing\_statistics\_8 = add\_form\_df(playing\_statistics\_8)

playing\_statistics\_9 = add\_form\_df(playing\_statistics\_9)

playing\_statistics\_10 = add\_form\_df(playing\_statistics\_10)

playing\_statistics\_11 = add\_form\_df(playing\_statistics\_11)

playing\_statistics\_12 = add\_form\_df(playing\_statistics\_12)

playing\_statistics\_13 = add\_form\_df(playing\_statistics\_13)

playing\_statistics\_14 = add\_form\_df(playing\_statistics\_14)

playing\_statistics\_15 = add\_form\_df(playing\_statistics\_15)

playing\_statistics\_16 = add\_form\_df(playing\_statistics\_16)

playing\_statistics\_17 = add\_form\_df(playing\_statistics\_17)

playing\_statistics\_18 = add\_form\_df(playing\_statistics\_18)

import pandas as pd

def rearrange\_columns(df, cols):

# Check if all required columns exist in the DataFrame

missing\_cols = [col for col in cols if col not in df.columns]

if not missing\_cols:

return df[cols]

else:

print(f"Warning: Missing columns {missing\_cols} in the DataFrame.")

return df

# List of columns to rearrange

cols = ['Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG', 'FTR', 'HTGS', 'ATGS', 'HTGC', 'ATGC', 'HTP', 'ATP', 'HM1', 'HM2', 'HM3',

'HM4', 'HM5', 'AM1', 'AM2', 'AM3', 'AM4', 'AM5' ]

# Apply rearrangement to each DataFrame

playing\_statistics\_1 = rearrange\_columns(playing\_statistics\_1, cols)

playing\_statistics\_2 = rearrange\_columns(playing\_statistics\_2, cols)

playing\_statistics\_3 = rearrange\_columns(playing\_statistics\_3, cols)

playing\_statistics\_4 = rearrange\_columns(playing\_statistics\_4, cols)

playing\_statistics\_5 = rearrange\_columns(playing\_statistics\_5, cols)

playing\_statistics\_6 = rearrange\_columns(playing\_statistics\_6, cols)

playing\_statistics\_7 = rearrange\_columns(playing\_statistics\_7, cols)

playing\_statistics\_8 = rearrange\_columns(playing\_statistics\_8, cols)

playing\_statistics\_9 = rearrange\_columns(playing\_statistics\_9, cols)

playing\_statistics\_10 = rearrange\_columns(playing\_statistics\_10, cols)

playing\_statistics\_11 = rearrange\_columns(playing\_statistics\_11, cols)

playing\_statistics\_12 = rearrange\_columns(playing\_statistics\_12, cols)

playing\_statistics\_13 = rearrange\_columns(playing\_statistics\_13, cols)

playing\_statistics\_14 = rearrange\_columns(playing\_statistics\_14, cols)

playing\_statistics\_15 = rearrange\_columns(playing\_statistics\_15, cols)

playing\_statistics\_16 = rearrange\_columns(playing\_statistics\_16, cols)

playing\_statistics\_17 = rearrange\_columns(playing\_statistics\_17, cols)

playing\_statistics\_18 = rearrange\_columns(playing\_statistics\_18, cols)

def get\_mw(playing\_stat):

j = 1

MatchWeek = []

for i in range(380):

MatchWeek.append(j)

if ((i + 1)% 10) == 0:

j = j + 1

playing\_stat['MW'] = MatchWeek

return playing\_stat

playing\_statistics\_1 = get\_mw(playing\_statistics\_1)

playing\_statistics\_2 = get\_mw(playing\_statistics\_2)

playing\_statistics\_3 = get\_mw(playing\_statistics\_3)

playing\_statistics\_4 = get\_mw(playing\_statistics\_4)

playing\_statistics\_5 = get\_mw(playing\_statistics\_5)

playing\_statistics\_6 = get\_mw(playing\_statistics\_6)

playing\_statistics\_7 = get\_mw(playing\_statistics\_7)

playing\_statistics\_8 = get\_mw(playing\_statistics\_8)

playing\_statistics\_9 = get\_mw(playing\_statistics\_9)

playing\_statistics\_10 = get\_mw(playing\_statistics\_10)

playing\_statistics\_11 = get\_mw(playing\_statistics\_11)

playing\_statistics\_12 = get\_mw(playing\_statistics\_12)

playing\_statistics\_13 = get\_mw(playing\_statistics\_13)

playing\_statistics\_14 = get\_mw(playing\_statistics\_14)

playing\_statistics\_15 = get\_mw(playing\_statistics\_15)

playing\_statistics\_16 = get\_mw(playing\_statistics\_16)

playing\_statistics\_17 = get\_mw(playing\_statistics\_17)

playing\_statistics\_18 = get\_mw(playing\_statistics\_18)

playing\_stat = pd.concat([playing\_statistics\_1,

playing\_statistics\_2,

playing\_statistics\_3,

playing\_statistics\_4,

playing\_statistics\_5,

playing\_statistics\_6,

playing\_statistics\_7,

playing\_statistics\_8,

playing\_statistics\_9,

playing\_statistics\_10,

playing\_statistics\_11,

playing\_statistics\_12,

playing\_statistics\_13,

playing\_statistics\_14,

playing\_statistics\_15,

playing\_statistics\_16,

playing\_statistics\_17,

playing\_statistics\_18

], ignore\_index=True)

# Gets the form points.

def get\_form\_points(string):

sum = 0

for letter in string:

sum += get\_points(letter)

return sum

playing\_stat['HTFormPtsStr'] = playing\_stat['HM1'] + playing\_stat['HM2'] + playing\_stat['HM3'] + playing\_stat['HM4'] + playing\_stat['HM5']

playing\_stat['ATFormPtsStr'] = playing\_stat['AM1'] + playing\_stat['AM2'] + playing\_stat['AM3'] + playing\_stat['AM4'] + playing\_stat['AM5']

playing\_stat['HTFormPts'] = playing\_stat['HTFormPtsStr'].apply(get\_form\_points)

playing\_stat['ATFormPts'] = playing\_stat['ATFormPtsStr'].apply(get\_form\_points)

# Identify Win/Loss Streaks if any.

def get\_3game\_ws(string):

if string[-3:] == 'WWW':

return 1

else:

return 0

def get\_5game\_ws(string):

if string == 'WWWWW':

return 1

else:

return 0

def get\_3game\_ls(string):

if string[-3:] == 'LLL':

return 1

else:

return 0

def get\_5game\_ls(string):

if string == 'LLLLL':

return 1

else:

return 0

playing\_stat['HTWinStreak3'] = playing\_stat['HTFormPtsStr'].apply(get\_3game\_ws)

playing\_stat['HTWinStreak5'] = playing\_stat['HTFormPtsStr'].apply(get\_5game\_ws)

playing\_stat['HTLossStreak3'] = playing\_stat['HTFormPtsStr'].apply(get\_3game\_ls)

playing\_stat['HTLossStreak5'] = playing\_stat['HTFormPtsStr'].apply(get\_5game\_ls)

playing\_stat['ATWinStreak3'] = playing\_stat['ATFormPtsStr'].apply(get\_3game\_ws)

playing\_stat['ATWinStreak5'] = playing\_stat['ATFormPtsStr'].apply(get\_5game\_ws)

playing\_stat['ATLossStreak3'] = playing\_stat['ATFormPtsStr'].apply(get\_3game\_ls)

playing\_stat['ATLossStreak5'] = playing\_stat['ATFormPtsStr'].apply(get\_5game\_ls)

playing\_stat.keys()

def only\_hw(string):

if string == 'H':

return 'H'

else:

return 'NH'

playing\_stat['FTR'] = playing\_stat.FTR.apply(only\_hw)

# Testing set (2015-16 season)

playing\_stat\_test = playing\_stat[5700:]

#saving the final dataset

playing\_stat.to\_csv('/content/Datasets/final\_dataset.csv')

#saving the test set

playing\_stat\_test.to\_csv("/content/Datasets/test.csv")

#loading the final dataset

dataset = pd.read\_csv('/content/Datasets/final\_dataset.csv')

dataset.head()

dataset.keys()

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

# Assuming 'dataset' is your DataFrame

# Select only numeric columns for correlation matrix calculation

numeric\_dataset = dataset.select\_dtypes(include=[float, int])

# Alternatively, if you want to drop non-numeric columns

# numeric\_dataset = dataset.drop(columns=['Date', 'HomeTeam', 'AwayTeam', 'FTR'])

# Compute the correlation matrix

correlation\_matrix = numeric\_dataset.corr()

# Set up the matplotlib figure

plt.figure(figsize=(20,10))

# Draw the heatmap

sns.heatmap(correlation\_matrix, annot=True, fmt='.2f', cmap='coolwarm', linewidths=0.5)

# Show the plot

plt.show()

# Print all column names in the dataset to verify their existence

print(dataset.columns)

# Now safely drop columns if they exist

columns\_to\_drop = ['Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG',

'HTGS', 'ATGS', 'HTGC', 'ATGC',

'HM4', 'HM5', 'AM4', 'AM5', 'MW', 'HTFormPtsStr',

'ATFormPtsStr', 'HTFormPts', 'ATFormPts', 'HTWinStreak3',

'HTWinStreak5', 'HTLossStreak3', 'HTLossStreak5', 'ATWinStreak3',

'ATWinStreak5', 'ATLossStreak3', 'ATLossStreak5', 'DiffPts']

# Drop only columns that exist in the dataset

columns\_to\_drop = [col for col in columns\_to\_drop if col in dataset.columns]

dataset2 = dataset.copy().drop(columns=columns\_to\_drop)

# Check the resulting dataframe

print(dataset2.head())

dataset2.keys()

dataset2.head(10)

#what is the win rate for the home team?

# Total number of matches.

n\_matches = dataset2.shape[0]

# Calculate number of features. -1 because we are saving one as the target variable (win/lose/draw)

n\_features = dataset2.shape[1] - 1

# Calculate matches won by home team.

n\_homewins = len(dataset2[dataset2.FTR == 'H'])

# Calculate win rate for home team.

win\_rate = (float(n\_homewins) / (n\_matches)) \* 100

# Print the results

print("Total number of matches: {}".format(n\_matches))

print ("Number of features: {}".format(n\_features))

print( "Number of matches won by home team: {}".format(n\_homewins))

print ("Win rate of home team: {:.2f}%".format(win\_rate))

import pandas as pd

# Convert last 3 wins columns to string type

X\_all['HM1'] = X\_all['HM1'].astype('str')

X\_all['HM2'] = X\_all['HM2'].astype('str')

X\_all['HM3'] = X\_all['HM3'].astype('str')

X\_all['AM1'] = X\_all['AM1'].astype('str')

X\_all['AM2'] = X\_all['AM2'].astype('str')

X\_all['AM3'] = X\_all['AM3'].astype('str')

# Function to preprocess features

def preprocess\_features(X):

''' Preprocesses the football data and converts categorical variables into dummy variables. '''

# Initialize new output DataFrame

output = pd.DataFrame(index = X.index)

# Investigate each feature column for the data

for col, col\_data in X.items(): # Use .items() instead of .iteritems()

# If data type is categorical, convert to dummy variables

if col\_data.dtype == object:

col\_data = pd.get\_dummies(col\_data, prefix=col)

# Collect the revised columns

output = output.join(col\_data)

return output

# Apply preprocessing

X\_all = preprocess\_features(X\_all)

print("Processed feature columns ({} total features):\n{}".format(len(X\_all.columns), list(X\_all.columns)))

X\_all.head(10)

from sklearn.model\_selection import train\_test\_split

# Shuffle and split the dataset into training and testing set.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_all, y\_all,

test\_size = 0.3,

random\_state = 2,

stratify = y\_all)

from sklearn.impute import SimpleImputer

from sklearn.linear\_model import LogisticRegression

from sklearn.pipeline import Pipeline

# Create an imputer object with a strategy (mean, median, most\_frequent, etc.)

imputer = SimpleImputer(strategy='mean')

# Create a pipeline that first imputes missing values, then applies Logistic Regression

pipeline = Pipeline([

('imputer', imputer),

('classifier', LogisticRegression(random\_state=0))

])

# Fit the pipeline on the training set

pipeline.fit(X\_train, y\_train)

from sklearn.impute import SimpleImputer

from sklearn.linear\_model import LogisticRegression

from sklearn.pipeline import Pipeline

# Define the columns with missing values

missing\_columns = ['HTP', 'ATP']

# Create a pipeline with an imputer and a logistic regression classifier

pipeline = Pipeline([

('imputer', SimpleImputer(strategy='mean')), # Impute missing values with the mean

('classifier', LogisticRegression(random\_state=0))

])

# Fit the pipeline on the training set

pipeline.fit(X\_train, y\_train)

# Optional: You can also transform the data to check the imputed values

X\_train\_imputed = pipeline.named\_steps['imputer'].transform(X\_train)

X\_test\_imputed = pipeline.named\_steps['imputer'].transform(X\_test)

# Import necessary libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from datetime import datetime as dt

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LogisticRegression

from sklearn.svm import SVC

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion\_matrix, classification\_report

from xgboost import XGBClassifier

from sklearn.model\_selection import GridSearchCV

from sklearn.preprocessing import scale

from sklearn.metrics import f1\_score, make\_scorer

# Extract relevant columns and concatenate data

columns\_req = ['Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG', 'FTR']

playing\_statistics = [df[columns\_req] for df in datasets]

# Function to calculate goals scored

def get\_goals\_scored(playing\_stat):

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

for i in range(len(playing\_stat)):

HTGS = playing\_stat.iloc[i]['FTHG']

ATGS = playing\_stat.iloc[i]['FTAG']

teams[playing\_stat.iloc[i].HomeTeam].append(HTGS)

teams[playing\_stat.iloc[i].AwayTeam].append(ATGS)

GoalsScored = pd.DataFrame(data=teams, index=[i for i in range(1, 39)]).T

GoalsScored[0] = 0

for i in range(2, 39):

GoalsScored[i] = GoalsScored[i] + GoalsScored[i-1]

return GoalsScored

# Function to calculate goals conceded

def get\_goals\_conceded(playing\_stat):

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

for i in range(len(playing\_stat)):

ATGC = playing\_stat.iloc[i]['FTHG']

HTGC = playing\_stat.iloc[i]['FTAG']

teams[playing\_stat.iloc[i].HomeTeam].append(HTGC)

teams[playing\_stat.iloc[i].AwayTeam].append(ATGC)

GoalsConceded = pd.DataFrame(data=teams, index=[i for i in range(1, 39)]).T

GoalsConceded[0] = 0

for i in range(2, 39):

GoalsConceded[i] = GoalsConceded[i] + GoalsConceded[i-1]

return GoalsConceded

# Function to get goals scored and conceded statistics

def get\_gss(playing\_stat):

GC = get\_goals\_conceded(playing\_stat)

GS = get\_goals\_scored(playing\_stat)

HTGS, ATGS, HTGC, ATGC = [], [], [], []

j = 0

for i in range(len(playing\_stat)):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

HTGS.append(GS.loc[ht][j])

ATGS.append(GS.loc[at][j])

HTGC.append(GC.loc[ht][j])

ATGC.append(GC.loc[at][j])

if ((i + 1) % 10) == 0:

j += 1

playing\_stat['HTGS'] = HTGS

playing\_stat['ATGS'] = ATGS

playing\_stat['HTGC'] = HTGC

playing\_stat['ATGC'] = ATGC

return playing\_stat

# Apply the function to each dataset

playing\_statistics = [get\_gss(df) for df in playing\_statistics]

# Function to calculate cumulative points

def get\_points(result):

if result == 'W':

return 3

elif result == 'D':

return 1

else:

return 0

def get\_cuml\_points(matchres):

matchres\_points = matchres.applymap(get\_points)

for i in range(2, 39):

matchres\_points[i] = matchres\_points[i] + matchres\_points[i-1]

matchres\_points.insert(column=0, loc=0, value=[0] \* len(matchres\_points))

return matchres\_points

def get\_matchres(playing\_stat):

teams = {team: [] for team in playing\_stat['HomeTeam'].unique()}

for i in range(len(playing\_stat)):

if playing\_stat.iloc[i].FTR == 'H':

teams[playing\_stat.iloc[i].HomeTeam].append('W')

teams[playing\_stat.iloc[i].AwayTeam].append('L')

elif playing\_stat.iloc[i].FTR == 'A':

teams[playing\_stat.iloc[i].AwayTeam].append('W')

teams[playing\_stat.iloc[i].HomeTeam].append('L')

else:

teams[playing\_stat.iloc[i].AwayTeam].append('D')

teams[playing\_stat.iloc[i].HomeTeam].append('D')

return pd.DataFrame(data=teams, index=[i for i in range(1, 39)]).T

def get\_agg\_points(playing\_stat):

matchres = get\_matchres(playing\_stat)

cum\_pts = get\_cuml\_points(matchres)

HTP, ATP = [], []

j = 0

for i in range(len(playing\_stat)):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

HTP.append(cum\_pts.loc[ht][j])

ATP.append(cum\_pts.loc[at][j])

if ((i + 1) % 10) == 0:

j += 1

playing\_stat['HTP'] = HTP

playing\_stat['ATP'] = ATP

return playing\_stat

# Apply the function to each dataset

playing\_statistics = [get\_agg\_points(df) for df in playing\_statistics]

# Function to get team form

def get\_form(playing\_stat, num):

form = get\_matchres(playing\_stat)

form\_final = form.copy()

for i in range(num, 39):

form\_final[i] = ''

j = 0

while j < num:

form\_final[i] += form[i-j]

j += 1

return form\_final

def add\_form(playing\_stat, num):

form = get\_form(playing\_stat, num)

h = ['M' for \_ in range(num \* 10)]

a = ['M' for \_ in range(num \* 10)]

j = num

for i in range(num \* 10, len(playing\_stat)):

ht = playing\_stat.iloc[i].HomeTeam

at = playing\_stat.iloc[i].AwayTeam

h.append(form.loc[ht][j][num-1])

a.append(form.loc[at][j][num-1])

if ((i + 1) % 10) == 0:

j += 1

playing\_stat[f'HM{num}'] = h

playing\_stat[f'AM{num}'] = a

return playing\_stat

def add\_form\_df(playing\_statistics):

for num in range(1, 6):

playing\_statistics = [add\_form(df, num) for df in playing\_statistics]

return playing\_statistics

# Add form data to datasets

playing\_statistics = add\_form\_df(playing\_statistics)

# Rearrange columns

cols = ['Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG', 'FTR', 'HTGS', 'ATGS', 'HTGC', 'ATGC', 'HTP', 'ATP',

'HM1', 'HM2', 'HM3', 'HM4', 'HM5', 'AM1', 'AM2', 'AM3', 'AM4', 'AM5']

playing\_statistics = [df[cols] for df in playing\_statistics]

# Function to add matchweek

def get\_mw(playing\_stat):

j = 1

MatchWeek = []

for i in range(len(playing\_stat)):

MatchWeek.append(j)

if ((i + 1) % 10) == 0:

j += 1

playing\_stat['MW'] = MatchWeek

return playing\_stat

# Add matchweek to datasets

playing\_statistics = [get\_mw(df) for df in playing\_statistics]

# Combine all data into a single DataFrame

playing\_stat = pd.concat(playing\_statistics, ignore\_index=True)

# Add form points

playing\_stat['HTFormPtsStr'] = playing\_stat['HM1'] + playing\_stat['HM2'] + playing\_stat['HM3'] + playing\_stat['HM4'] + playing\_stat['HM5']

playing\_stat['ATFormPtsStr'] = playing\_stat['AM1'] + playing\_stat['AM2'] + playing\_stat['AM3'] + playing\_stat['AM4'] + playing\_stat['AM5']

playing\_stat['HTFormPts'] = playing\_stat['HTFormPtsStr'].apply(get\_form\_points)

playing\_stat['ATFormPts'] = playing\_stat['ATFormPtsStr'].apply(get\_form\_points)

# Identify Win/Loss Streaks

playing\_stat['HTWinStreak3'] = playing\_stat['HTFormPtsStr'].apply(get\_3game\_ws)

playing\_stat['HTWinStreak5'] = playing\_stat['HTFormPtsStr'].apply(get\_5game\_ws)

playing\_stat['HTLossStreak3'] = playing\_stat['HTFormPtsStr'].apply(get\_3game\_ls)

playing\_stat['HTLossStreak5'] = playing\_stat['HTFormPtsStr'].apply(get\_5game\_ls)

playing\_stat['ATWinStreak3'] = playing\_stat['ATFormPtsStr'].apply(get\_3game\_ws)

playing\_stat['ATWinStreak5'] = playing\_stat['ATFormPtsStr'].apply(get\_5game\_ws)

playing\_stat['ATLossStreak3'] = playing\_stat['ATFormPtsStr'].apply(get\_3game\_ls)

playing\_stat['ATLossStreak5'] = playing\_stat['ATFormPtsStr'].apply(get\_5game\_ls)

# Calculate Goal Difference

playing\_stat['HTGD'] = playing\_stat['HTGS'] - playing\_stat['HTGC']

playing\_stat['ATGD'] = playing\_stat['ATGS'] - playing\_stat['ATGC']

# Calculate DiffPts and DiffFormPts

playing\_stat['DiffPts'] = playing\_stat['HTP'] - playing\_stat['ATP']

playing\_stat['DiffFormPts'] = playing\_stat['HTFormPts'] - playing\_stat['ATFormPts']

# Scale features by Matchweek

playing\_stat.MW = playing\_stat.MW.astype(float)

cols = ['HTGD', 'ATGD', 'DiffPts', 'DiffFormPts', 'HTP', 'ATP']

for col in cols:

playing\_stat[col] = playing\_stat[col] / playing\_stat.MW

# Transform target variable

def only\_hw(string):

if string == 'H':

return 'H'

else:

return 'NH'

playing\_stat['FTR'] = playing\_stat.FTR.apply(only\_hw)

# Save final dataset

playing\_stat.to\_csv('/content/final\_dataset.csv', index=False)

# Load the dataset

dataset = pd.read\_csv('/content/final\_dataset.csv')

# Drop columns to prevent multicollinearity

dataset2 = dataset.copy().drop(columns=['Date', 'HomeTeam', 'AwayTeam', 'FTHG', 'FTAG', 'HTGS', 'ATGS', 'HTGC', 'ATGC',

'HM4', 'HM5', 'AM4', 'AM5', 'MW', 'HTFormPtsStr', 'ATFormPtsStr',

'HTWinStreak3', 'HTWinStreak5', 'HTLossStreak3', 'HTLossStreak5',

'ATWinStreak3', 'ATWinStreak5', 'ATLossStreak3', 'ATLossStreak5',

'DiffPts'])

# Split the dataset into features and target variable

X\_all = dataset2.drop(['FTR'], axis=1)

y\_all = dataset2['FTR']

# Standardizing numerical features

cols = [['HTGD', 'ATGD', 'HTP', 'ATP']]

for col in cols:

X\_all[col] = scale(X\_all[col])

# Convert categorical features to dummy variables

X\_all = pd.get\_dummies(X\_all, columns=['HM1', 'HM2', 'HM3', 'AM1', 'AM2', 'AM3'], drop\_first=True)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_all, y\_all, test\_size=0.3, random\_state=2, stratify=y\_all)

# Function to evaluate model performance

def evaluate\_model(model, X\_test, y\_test):

y\_pred = model.predict(X\_test)

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt='d')

plt.show()

print(classification\_report(y\_test, y\_pred))

from sklearn.preprocessing import LabelEncoder

# Encode target labels with value between 0 and n\_classes-1

label\_encoder = LabelEncoder()

y\_train\_encoded = label\_encoder.fit\_transform(y\_train)

y\_test\_encoded = label\_encoder.transform(y\_test)

# Logistic Regression

lr\_model = LogisticRegression(random\_state=0)

lr\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model(lr\_model, X\_test, y\_test\_encoded)

# SVM

svm\_model = SVC(kernel='rbf', random\_state=0)

svm\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model(svm\_model, X\_test, y\_test\_encoded)

# Random Forest

rf\_model = RandomForestClassifier(criterion='gini', n\_estimators=700, min\_samples\_split=10, min\_samples\_leaf=1,

max\_features='sqrt', oob\_score=True, random\_state=1, n\_jobs=-1)

rf\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model(rf\_model, X\_test, y\_test\_encoded)

# XGBoost

xgb\_model = XGBClassifier(seed=82)

xgb\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model(xgb\_model, X\_test, y\_test\_encoded)

# Hyperparameter Tuning for XGBoost

parameters = {

'learning\_rate': [0.1],

'n\_estimators': [40],

'max\_depth': [3],

'min\_child\_weight': [3],

'gamma': [0.4],

'subsample': [0.8],

'colsample\_bytree': [0.8],

'scale\_pos\_weight': [1],

'reg\_alpha': [1e-5]

}

f1\_scorer = make\_scorer(f1\_score, pos\_label=1)

grid\_obj = GridSearchCV(XGBClassifier(seed=2), param\_grid=parameters, scoring=f1\_scorer, cv=5)

grid\_obj = grid\_obj.fit(X\_train, y\_train\_encoded)

best\_xgb = grid\_obj.best\_estimator\_

print("Best XGBoost model:", best\_xgb)

# Evaluate the best model

evaluate\_model(best\_xgb, X\_test, y\_test\_encoded)

from sklearn.preprocessing import LabelEncoder

from sklearn.ensemble import GradientBoostingClassifier

from sklearn.metrics import mean\_absolute\_error, r2\_score, mean\_absolute\_percentage\_error

# Function to evaluate model performance with additional metrics

def evaluate\_model\_with\_metrics(model, X\_test, y\_test):

y\_pred = model.predict(X\_test)

cm = confusion\_matrix(y\_test, y\_pred)

sns.heatmap(cm, annot=True, fmt='d')

plt.show()

print(classification\_report(y\_test, y\_pred))

# Calculate additional metrics

mae = mean\_absolute\_error(y\_test, y\_pred)

mape = mean\_absolute\_percentage\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Mean Absolute Error (MAE): {mae}")

print(f"Mean Absolute Percentage Error (MAPE): {mape}")

print(f"R-squared (R²): {r2}")

# Encode target labels with value between 0 and n\_classes-1

label\_encoder = LabelEncoder()

y\_train\_encoded = label\_encoder.fit\_transform(y\_train)

y\_test\_encoded = label\_encoder.transform(y\_test)

# Logistic Regression

lr\_model = LogisticRegression(random\_state=0)

lr\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model\_with\_metrics(lr\_model, X\_test, y\_test\_encoded)

# SVM

svm\_model = SVC(kernel='rbf', random\_state=0)

svm\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model\_with\_metrics(svm\_model, X\_test, y\_test\_encoded)

# Random Forest

rf\_model = RandomForestClassifier(criterion='gini', n\_estimators=700, min\_samples\_split=10, min\_samples\_leaf=1,

max\_features='sqrt', oob\_score=True, random\_state=1, n\_jobs=-1)

rf\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model\_with\_metrics(rf\_model, X\_test, y\_test\_encoded)

# XGBoost

xgb\_model = XGBClassifier(seed=82)

xgb\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model\_with\_metrics(xgb\_model, X\_test, y\_test\_encoded)

# Gradient Boosting Machines (GBM)

gbm\_model = GradientBoostingClassifier(random\_state=0)

gbm\_model.fit(X\_train, y\_train\_encoded)

evaluate\_model\_with\_metrics(gbm\_model, X\_test, y\_test\_encoded)

# Hyperparameter Tuning for XGBoost

parameters = {

'learning\_rate': [0.1],

'n\_estimators': [40],

'max\_depth': [3],

'min\_child\_weight': [3],

'gamma': [0.4],

'subsample': [0.8],

'colsample\_bytree': [0.8],

'scale\_pos\_weight': [1],

'reg\_alpha': [1e-5]

}

f1\_scorer = make\_scorer(f1\_score, pos\_label=1)

grid\_obj = GridSearchCV(XGBClassifier(seed=2), param\_grid=parameters, scoring=f1\_scorer, cv=5)

grid\_obj = grid\_obj.fit(X\_train, y\_train\_encoded)

best\_xgb = grid\_obj.best\_estimator\_

print("Best XGBoost model:", best\_xgb)

# Evaluate the best XGBoost model

evaluate\_model\_with\_metrics(best\_xgb, X\_test, y\_test\_encoded)

from sklearn.model\_selection import RandomizedSearchCV

import numpy as np

# Define the parameter grid for each model with reduced search space

param\_grid\_rf = {

'n\_estimators': [100, 200],

'max\_depth': [10, None],

'min\_samples\_split': [2, 10],

'min\_samples\_leaf': [1, 2],

'max\_features': ['sqrt', 'log2']

}

param\_grid\_gbm = {

'n\_estimators': [100, 200],

'learning\_rate': [0.01, 0.1],

'max\_depth': [3, 4],

'subsample': [0.8, 1.0],

'min\_samples\_split': [2, 5],

'min\_samples\_leaf': [1, 2]

}

param\_grid\_xgb = {

'learning\_rate': [0.1, 0.05],

'n\_estimators': [40, 100],

'max\_depth': [3, 4],

'min\_child\_weight': [1, 3],

'gamma': [0.1, 0.3],

'subsample': [0.8, 1.0],

'colsample\_bytree': [0.8, 1.0],

'reg\_alpha': [1e-5, 1e-2]

}

# Hyperparameter tuning for Random Forest using RandomizedSearchCV

grid\_rf = RandomizedSearchCV(RandomForestClassifier(random\_state=1), param\_distributions=param\_grid\_rf,

scoring=f1\_scorer, cv=3, n\_iter=10, n\_jobs=-1, random\_state=1)

grid\_rf.fit(X\_train, y\_train\_encoded)

best\_rf = grid\_rf.best\_estimator\_

print("Best Random Forest model:", best\_rf)

evaluate\_model\_with\_metrics(best\_rf, X\_test, y\_test\_encoded)

# Hyperparameter tuning for Gradient Boosting Machines (GBM) using RandomizedSearchCV

grid\_gbm = RandomizedSearchCV(GradientBoostingClassifier(random\_state=0), param\_distributions=param\_grid\_gbm,

scoring=f1\_scorer, cv=3, n\_iter=10, n\_jobs=-1, random\_state=1)

grid\_gbm.fit(X\_train, y\_train\_encoded)

best\_gbm = grid\_gbm.best\_estimator\_

print("Best GBM model:", best\_gbm)

evaluate\_model\_with\_metrics(best\_gbm, X\_test, y\_test\_encoded)

# Hyperparameter tuning for XGBoost using RandomizedSearchCV

grid\_xgb = RandomizedSearchCV(XGBClassifier(seed=2), param\_distributions=param\_grid\_xgb,

scoring=f1\_scorer, cv=3, n\_iter=10, n\_jobs=-1, random\_state=1)

grid\_xgb.fit(X\_train, y\_train\_encoded)

best\_xgb = grid\_xgb.best\_estimator\_

print("Best XGBoost model:", best\_xgb)

evaluate\_model\_with\_metrics(best\_xgb, X\_test, y\_test\_encoded)